Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT

Michael B. Jones
John Regehr

July, 1998

Technical Report
MSR-TR-98-29

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Paper published in Proceedings of the Eighth International Workshop on
Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98),
Cambridge, England, pages 107-110, July 1998.

UU IR Author Manuscript

Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9s/1
Redmond, WA 98052, USA

mbj@microsoft.com
http://research.microsoft.com/~mbj/

John Regehr*

Department of Computer Science, Thornton Hall
University of Virginia
Charlottesville, VA 22903-2242, USA

regehr@virginia.edu
http://www.cs.virginia.edu/~jdr8d/

Abstract

This paper presents a snapshot of early results from a study of Windows NT aimed at understanding and improving its limitations when used for time-dependent tasks, such as those that arise for audio and video processing.

Clearly there are time scales for which it can achieve effectively perfect reliability, such as the one-second deadlines present in the Tiger Video Filesystem. Other time scales, such as reliable sub-millisecond scheduling of periodic tasks in user space, are clearly out of reach. Yet, there is an interesting middle ground between these time scales in which deadlines may be met, but will not always be. This study focuses on system and application behaviors in this region with the short-term goals of understanding and improving the real-time responsiveness of applications using Windows NT 5.0 and a longer-term goal of prototyping and recommending possible scheduling and resource management enhancements to future Microsoft systems products.

Finally, while this paper primarily contains examples and results from Windows NT, we believe that the kinds of limitations and artifacts identified may also apply to other commodity systems such as the many UNIX variants. Indeed, this paper is primarily intended to provide a starting point for fruitful discussions along these lines at the workshop and not as a record of completed work.

1. Introduction

Windows NT and other commonly available general-purpose operating systems such as Solaris and Linux are increasingly being used to run time-dependent tasks such as those that arise for audio and video processing, despite good arguments against doing so [Nieh et al. 93]. This is the case even though many such systems, and Windows NT [Solomon 98] in particular, were designed primarily to maximize aggregate throughput and to achieve approximately fair sharing of resources rather than to provide low-latency response to events, predictable time-based scheduling, or explicit resource allocation mechanisms. Nonetheless, since these systems are being used for time-dependent tasks, it is important to understand both their capabilities and limitations for such applications.

We are in the early stages of a study of Windows NT aimed at understanding and improving its limitations when used for time-dependent tasks. Clearly there are time scales for which it can achieve effectively perfect reliability, such as for the one-second deadlines present in the Tiger Video Filesystem [Bolosky et al. 97]. Other time scales, such as reliable sub-millisecond scheduling of periodic tasks, are clearly out of reach without resorting to special tricks. But there is an interesting range in the middle where tasks can often be executed, but not always. Many practical multimedia activities, such as fine-grained real-time audio synthesis, fall into this middle range.

This paper is intended to serve as a catalyst for discussions on the effectiveness of and problems with using commodity operating systems for time-dependent tasks. While it does provide a snapshot of some of the early findings from our study of Windows NT, it is not a record of completed work. Rather, it is intended to provide some concrete starting points for the discussion at the workshop based on real data.

2. Windows NT Background

Windows NT contains few of the systems facilities and design features that are commonly accepted as providing effective underpinnings for time-dependent applications. Features not found include deadline-based scheduling, explicit CPU or resource management [Mercer et al. 94, Nieh & Lam 97, Jones et al. 97], priority inheritance, fine-granularity clock and timer services [Jones et al. 96], and bounded response time for essential system services. Features that it does have include high-priority real-time thread priorities, interrupt routines that typically re-enable interrupts very quickly, and periodic callback routines [Solomon 98].

Windows NT schedules threads based on their priority and processor affinity. The priorities are divided into three ranges: real-time (16-31), normal (1-15), and idle (0). Priorities of threads in the normal range are boosted following I/O completions and decreased when the thread's time quantum runs out, as is often done in time-sharing systems. The system never adjusts the priorities of threads in the real-time range. The scheduler essentially selects the first thread of the highest runnable

* John Regehr is a research intern at Microsoft Research during the summer of 1998.

UU IR Author Manuscript

priority and runs it for its quantum, then places it at the tail of its priority list. For more details on processor affinity and related issues, see [Solomon 98].

Under Windows NT, not all CPU time is controlled by the scheduler. Of course, time spent in interrupt handling is unscheduled, although the system is designed to minimize hardware interrupt latencies by doing as little work as possible at interrupt level. Instead, much driver-related work occurs in *Deferred Procedure Calls* (DPCs). DPCs are routines executed within the kernel in no particular thread context in response to queued requests for their execution. For example, DPCs check the timer queues for expired timers and process the completion of I/O requests. Hardware interrupt latency is reduced by having interrupt handlers queue DPCs to finish the work associated with them. All queued DPCs are executed whenever a thread is selected for execution just prior to starting the selected thread. While good for interrupt latencies, DPCs can be bad for thread scheduling latencies, as they can potentially result in an unbounded amount of work running before a thread is scheduled.

To enhance system portability, Windows NT uses a loadable *Hardware Abstraction Layer* (HAL) module that isolates the kernel and drivers from low-level hardware details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms. The system clock is one service provided by each HAL. The HAL generates periodic clock interrupts for the kernel. The HAL interface contains no means of requesting a single interrupt at a particular time.

The Win32 interface contains a facility called *Multimedia Timers* supporting periodic execution of application code at a frequency specified by the application. The period is specified in 1ms increments. By default, the Windows NT kernel receives a clock interrupt every 10 to 15ms; to permit more accurate timing, multimedia timers internally use a Windows NT system call that allows the timer interrupt frequency to be adjusted within the range permitted by the HAL (typically 1-15ms).

Multimedia timers are implemented by spawning a high-priority thread that sets a kernel timer and then blocks. Upon awakening the thread executes a callback routine provided by the user, schedules its next wakeup, and then goes back to sleep.

3. Baseline Performance Measurements

Given that Multimedia Timers are the primary mechanism available for applications to request timely execution of code, it is important for time-sensitive applications to understand how well it works in practice. We designed a set of experiments to determine this.

We wrote a test application that sets the clock frequency to the smallest period supported by the HAL (~1ms for all HALs used in these tests) and requests callbacks every 1ms. Each callback just records the Pentium cycle counter value at which it occurred in pinned memory. The application runs at the highest real-time priority. Note that the application is blocked waiting for a callback nearly 100% of the time, and so imposes no significant load on the system. The core of the application is as follows:

```
int main(…) {
    timeGetDevCaps(&TimeCap, …);
    timeBeginPeriod(TimeCap.wPeriodMin);
        // Set clock period to min supported
    TimerID = timeSetEvent(
        // Start periodic callback
        1,  // period (in milliseconds)
        0,  // resolution (0 = maximum)
        CallBack,   // callback function
        0,  // no user data
        TIME_PERIODIC);  // periodic timer
}
void Callback(…) {
    TimeStamp [i++] = ReadTimeStamp();
        // Record Pentium cycle counter value
}
```

On an ideal computer system dedicated to this program the callbacks would occur exactly 1ms apart. Actual runs allow us to determine how close real versions of Windows NT running on real hardware come to this.

Performance measurements were made on two different machines:

- a Pentium Pro 200MHz uniprocessor, with both an Intel EtherExpress 16 ISA Ethernet card and a DEC 21140 DC21x4-based PCI Fast Ethernet card, running uniprocessor kernels, using the standard uniprocessor PC HAL, HALX86.
- a Pentium 2 333MHz uniprocessor (but with a dual-processor motherboard) with an Intel EtherExpress Pro PCI Ethernet card, running multiprocessor kernels, using the standard multiprocessor PC HAL, HALMPS.

NT4 measurements were made under Windows NT 4.0, Service Pack 3. NT5 measurements were made under Windows NT 5.0, build 1805 (a developer build between Beta 1 and Beta 2). All measurements were made while attached to the network.

3.1 Supported Clock Rates

The standard uniprocessor HAL advertises support for clock rates in the range 1003µs to 14995µs. The actual rate observed during our tests was equal to the minimum, 1003µs. This was true for both NT4 and NT5.

The standard multiprocessor HAL advertises support for clock rates in the range 1000µs to 15625µs. The actual rate observed during our tests, however, was 976µs – less than the advertised minimum. See Section 4.1 for some of the implications of this fact. Once again, these observations were consistent across NT4 and NT5.

Finally, note that some HALs do not even support variable clock rates. This limits Multimedia Timer resolution to a constant clock rate chosen by the HAL.

UU IR Author Manuscript

3.2 Times Between Timer Callbacks

Table 1 gives statistics for typical 10-second runs of the test application on both test machines for both operating system versions.

Times Between Callbacks	PPro, NT4	PPro, NT5	P2, NT4	P2, NT5
Minimum µs	31	31	20	33
Maximum µs	2384	18114	2144	2396
Average µs	999	999	999	999
Std Dev µs	70	211	955	941

Table 1: Statistics about Times Between Callbacks

All provide an average time between callbacks of 999µs, but the similarities end there. Note, for instance, that the standard deviation for the Pentium 2 runs is around 950µs – nearly equal to the mean! Also, notice that there was at least one instance on the Pentium Pro under NT5 when no callback occurred for over 18ms.

The statistics do not come close to telling the full story. Table 2 is a histogram of the actual times between callbacks for these same runs, quantized into 100µs bins.

# Times Between Callbacks Falling Within Interval	PPro, NT4	PPro, NT5	P2, NT4	P2, NT5
0-100µs	34	62	4880	4880
100-200µs	1			
300-400µs		1		
500-600µs	4	2		
600-700µs	6	1		
700-800µs	22			
800-900µs	150	10		
900-1000µs	571	1281		
1000-1100µs	9014	8627		
1100-1200µs	161	10		
1200-1300µs	28	1		
1300-1400µs	6	1		
1400-1500µs	1	1		2
1700-1800µs			2	5
1800-1900µs			9	91
1900-2000µs			5107	5014
2000-2100µs				4
2100-2200µs			2	2
2300-2400µs	2			2
7700-7800µs		2		
18100-18200µs		1		

Table 2: Histogram of Times Between Callbacks

Now, the reason for the high standard deviation for the Pentium 2 runs is clear – no callbacks occurred with spacings anywhere close to the desired 1ms apart. Instead, about half occurred close to 0ms apart and half occurred about 2ms apart!

Also, for the Pentium Pro NT5 run, note that twice callbacks occurred about 7.7ms apart and once over 18ms apart. In fact, this is not atypical. On this configuration, there are *always* two samples around 7-8ms apart and one around 18ms apart.

Indeed, the point of our study is to try to learn what is causing anomalies such as these, and to fix them!

4. Early Results

This section presents two snapshots of the kinds of problems we discovered (and one of which we already fixed!) during preliminary latency debugging of applications using Multimedia Timers.

4.1 HAL Timing Differences

Because the HAL virtualizes the hardware timer interface, HAL writers may implement timers in different ways. For example, HALX86 uses the 8254 clock chip to generate clock interrupts on IRQ1, but HALMPS uses the Real Time Clock (RTC) to generate interrupts on IRQ8.

Upon receiving a clock interrupt, the HAL calls up to the Windows NT kernel, which (among other things) compares the current time to the expiration time of any pending timers, and dequeues and processes those timers whose expiration times have passed.

As we have seen, multimedia timers are able to meet 1ms deadlines most of the time on machines running HALX86. To understand why 1ms timers do not work on machines running HALMPS, we next examine the timer implementation in more detail.

A periodic multimedia timer always knows the time at which it should next fire; every time it does fire, it increments this value by the timer interval. If the next firing time is ever in the past, the timer repeatedly fires until the time has moved into the future. The next firing time is rounded to the nearest millisecond. This interacts poorly with HALMPS, which approximates 1ms clock interrupts by firing at 1024Hz, or every 976µs. (The RTC only supports power-of-2 frequencies.)

Because the interrupt frequency is slightly higher than the timer frequency, we would expect to occasionally wait almost 2ms for a callback when the 976µs interrupt interval happens to be contained within the 1000µs timer interval. Unfortunately, rounding the firing time ensures that this worst case becomes the common case. Since it never asks to wait less than 1ms, it always waits nearly 2ms before expiring, then fires again immediately to catch up, hence the observed behavior.

We fixed this error by modifying the timer implementation to compute the next firing time more precisely, allowing it to request wakeups less than 1ms in the future. (An alternative fix would have been to use periodic kernel timers, rather than repeatedly setting one-shot timers.) The results of our fix can be seen in Table 3.

As expected, approximately 2.4% of the wakeups occur near 2ms, since clock interrupts arrive 2.4% faster than timers. As a number of HALs besides HALMPS use the RTC, this fix should be generally useful.

# Times Between Callbacks Falling Within Interval	**P2, NT5**	**P2, NT5 fixed**
0-100µs	4880	1
500-600µs		1
600-700µs		3
700-800µs		2
800-900µs		7
900-1000µs		9609
1000-1100µs		127
1100-1200µs		4
1200-1300µs		2
1300-1400µs		2
1400-1500µs	2	2
1700-1800µs	5	
1800-1900µs	91	
1900-2000µs	5014	240
2000-2100µs	4	
2100-2200µs	2	
2300-2400µs	2	

Table 3: Histogram Showing Results of Timer Fix

4.2 Long-Running DPCs

DPCs are not scheduled by the thread scheduler; they take precedence over any thread. Therefore, long-running DPCs are an obstacle to precise scheduling of user code.

At the time of this writing, we can definitely attribute the 18.1ms and 7.7ms delays on the Pentium Pro to DPC activity. We know, for instance, that the 7ms DPC is in dc21x4.sys – the driver for the DEC Fast Ethernet card. And the 18ms are spent within the NDIS subsystem in a DPC that is intended to check for stalled miniport drivers. Both of these DPCs bear further study in order to identify the exact problem and propose a solution.

5. Methodology

Our primary method of discovering and diagnosing timing problems is to produce instrumented versions of applications, the kernel, and relevant drivers that record timing information in physical memory buffers. After runs in which interesting anomalies occur, a combination of perl scripts and human eyeballing are used to condense and correlate the voluminous timing logs to extract the relevant bits of information from them.

Typically, after a successful run and log analysis, the conclusion is that more data is needed to understand the behavior. So additional instrumentation is added, usually to the kernel, thus unfortunately the edit/compile/debug cycle often gets a reboot step added to it. This approach works but we would be open to ways to improve it.

6. Future Work

Improving predictability of the existing Windows NT features used by time-dependent programs is clearly important, but without better scheduling and resource management support, this can only help so much. In addition to continuing to study and improve the real-time performance of the existing features, we also plan to prototype better underpinnings for real-time applications.

7. Conclusions

While the essential structure of Windows NT is capable of providing low-latency response to events, obvious (and often easy to fix!) problems we have seen, such as the poor interaction between multimedia timers and HALMPS and occasional long DPC execution times, keep current versions of Windows NT from guaranteeing timely response to real-time events below thresholds in the tens of milliseconds. Bottom line – the system is clearly not being actively developed or tested for real-time responsiveness. We are working to change that!

While the details of this paper are obviously drawn from Windows NT, we believe that similar problems for time-dependent tasks will also be found in other general-purpose commodity systems for similar reasons. We look forward to discussing this at the workshop.

Acknowledgments

The authors wish to thank Patricia Jones for her editorial assistance in the preparation of this manuscript.

References

[Bolosky et al. 97] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed Schedule Management in the Tiger Video Fileserver. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, St-Malo, France, pp. 212-223, Oct. 1997.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Roşu, Marcel-Cătălin Roşu. An Overview of the Rialto Real-Time Architecture. In *Proc. of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 249-256, Sep. 1996.

[Jones et al. 97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu, CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities, In *Proc. of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pp. 198-211, Oct. 1997.

[Mercer et al. 94] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Nieh et al. 93] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerald Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proc. of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*. Lancaster, U.K., Nov. 1993.

[Nieh & Lam 97] Jason Nieh and Monica S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, St-Malo, France, pp. 184-197, Oct. 1997.

[Solomon 98] David A. Solomon. *Inside Windows NT, Second Edition.* Microsoft Press, 1998.